Algorithmic Analysis of the Impact of Changes on Object-Oriented Software

Li Li
LCC L.L.C.
2300 Clarendon Blvd., Suite 800
Arlington, VA 22201
phone: 703-516-7394
email: lili@lccinc.com

A. Jefferson Offutt
ISSE Department, 4A4
George Mason University
Fairfax, VA 22030-4444
phone: 703-993-1654
email: ofut@isse.gmu.edu

GMU ISSE Technical Report ISSE-TR-96-02

abstract

As software ages and evolves, tasks of maintaining it become more complex and more expensive. Without change impact analysis, engineers could make critical changes in the dark that could cause major problems or big ripple effects in the system. In this paper, we analyze the possible changes that could happen in object-oriented software, how these changes affect other classes in the system, and describes a set of algorithms that can find out all the possibly affected classes if these changes happened. This technique allows software developers to perform "what if" analysis on the impact of proposed process changes in the object-oriented system, choose the proposed change that causes the minimum impact on the system. Once the change is committed, it allows software testers to know what the areas are in the software system that are possibly affected by the change and retest only those classes instead of whole system and still feel confident about the software.

Key words: Change Impact Analysis, Object-Oriented Software, Software Testing.

1. Introduction

The software systems have traditionally been decomposed into subsystems top down according to their functionality. The object-oriented approach describes the system in terms of objects that make up the problem domain. Applying object-oriented technology can lead to better system architectures, and enforces a disciplined coding style. Rumbaugh [Rum91] states that an object-oriented approach produces a clean, well-understood design that is easier to test, maintain, and extend than non-object-oriented designs because the object classes provide a natural unit of modularity.

Despite the advantages of object-oriented technology, it does not by itself ensure the quality of the software, shield against developer's mistakes, nor prevent faults. Barbey and Strohmeier think the object-oriented paradigm can also be a hindrance to testing, due to encapsulation, inheritance, and polymorphism[Bar94].

As time goes by, there are more demands for evolving existing software. Software evolution refers to the on-going enhancements of existing software systems, involving both development and maintenance. As software ages and evolves, the task of maintaining it becomes more complex and more expensive, which is especially true for systems implemented in object-oriented approach.

When engineers consider an update to an existing system, they need to know what potential impacts this update will bring to the entire system. Determining how a potential change might impact the system is refereed to as *change impact analysis*. Without impact analysis, engineers could make critical changes that could cause major problems or have ripple effects throughout the system. To predict impact of changes to object-oriented software, users can use the technique described in this paper to evaluate the effects of changes before commit them to the system. During maintenance, when

some changes have been made to the system, we need to estimate how many classes need to be retested. Retesting too many classes in the system will increase the cost of testing, retesting too few classes in the system might adversely effect the quality of the software. By applying this technique, testers can learn what classes are possibly affected by the change and retest only those classes.

In this paper, we analyze a number of possible changes to object-oriented software, how these changes affect the classes in the system, and describe a set of algorithms that determin what classes will be affected by the changes.

Section 2 presents object-oriented concepts and definitions used in the paper and describes the theoretical background of our algorithms. Section 4 first analyzes how the encapsulation, inheritance, polymorphism will affect the change propagation, and then describes a simple algorithm to estimate potentially affected classes. Algorithms are presented that calculate change propagation within classes, between client and server classes, and between parent and children classes. We also categorize the possible changes that could happen to the system and give each type of change a change attribute according to how these different types of changes can affect the other parts of the system, and discuss how to optimize the original algorithms according to these different change categories. An example system a given, and the algorithms described in this paper are applied to the example system to analyze the impact result. The complete set of algorithms are given in the Technical Report[xxx]. Section 6 is the conclusion, it describes the plan we are interesting in.

2. Definition

An object-oriented system is composed of objects and classes. An object is composed of a set of *properties*, which define its state, and a set of *operations*, which define its behavior. The *state* of an object encompasses all the properties of the object plus the current values of each of these properties. *Behavior* is how an object acts and reacts, in terms of its state changes and message passing[Boo94]. The state of an object represents the cumulative results of its behavior. The constants and variables that serves as the representation of its instance's state can be called *Fields*, *Instance Variables* or *Data members* depend on the language. They are used interchangeably in this paper. *Messages* are operations that one object performs upon another, *Methods* or *Member Function* are operations that clients may perform upon an object. A *Class* is the specification of an object; it is the "blueprint" from which an object can be created. A class describes an object's interface, the structure of its state information, and the details of its methods [Mar95]. Objects are runtime instances of a class. An *Abstract Class* is a class that only partially describes an object. Usually some or all of its interfaces are without implementation.

A *control flow graph* (CFG) is a finite, connected directed graph $G = (N, E, N_s, N_f)$ where N is a finite set of nodes, $E \subseteq NxN$ is a finite set of edges, $N_s \in N$ is the start node and $N_f \in N$ is the final node. A node in a CFG represents a statement or a basic block, i.e. a sequence of statements having the property that each statement in the sequence is executed whenever the first statement is executed. An edge (Ni, Nj) represents a possible flow of control between two statements or basic blocks, i.e. the statement (or block) represented by Ni is executed before the statement or basic block that is represented by Nj.

A *Data Definition* is an expression or part of an expression that modifies a data item. A *Data Use* is an expression or that part of an expression that references a data item without modifying it. A *def-use* pair is a definition that may, under some executions, reach the use without going through another definition. A *data flow graph* (Def-Use) graph is a directed graph where the nodes and some edges are described by def_use relationships.

The *reflective transitive closure* of a relationship R is the relation R^+ defined by c R^+ d, if and only if there is a sequence $e_1 R e_2, e_2 R e_3, ..., e_{m-1} R e_m$, Where m $\geq 2, c = e_1, and\ d = e_m$.

In this paper, we use Booch Notation [Boo94] to express relationships among classes. Figure 1, "Some Booch notation for class diagram" shows Booch Notations that express some class relations used in this paper.

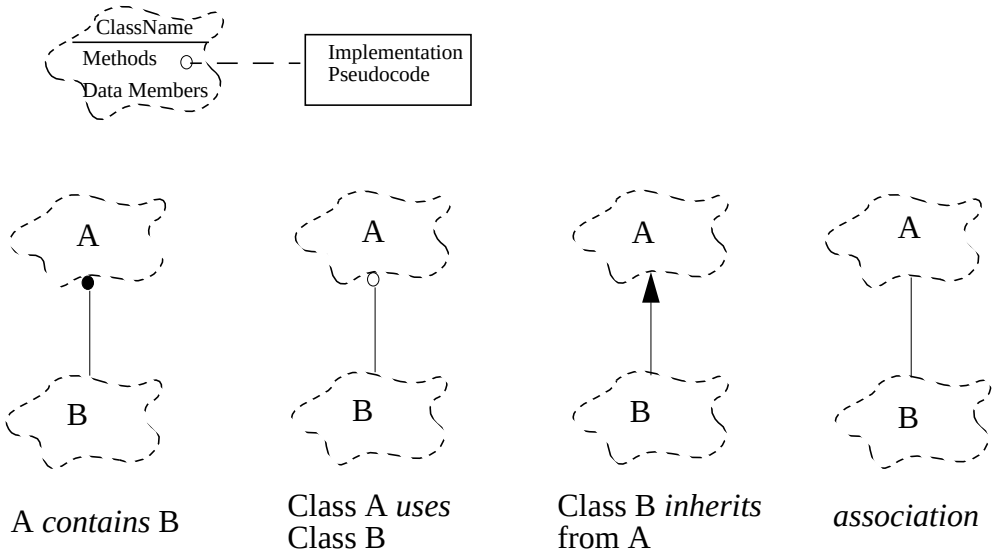


Figure 1: Some Booch notation for class diagrams

Class A *contains* class B if the instance of class B is held in one of the instance variables of the A. This represents the "whole/part" relationship. For example, we can say a car has an engine, or a car has doors. Class A *uses* class B if A sends messages to B. For example, we say a person uses a car. The person tells the car to start-up, turn, stop by sending messages to the car through car interface like key, steering etc. A class can *inherit* the instance variables, interfaces, and instance methods of another class as if they were defined within it. This expresses the generalization/specialization relationship. For example, a Sedan is a specialization of a general car. The class from which another class inherits is called *parent* or *superclass*. The class that inherits from parent is called a *child*, *subclass* or *derived class*. If a class has more than one parent, this kind of relationship is called *multiple inheritance*. *Association* is a semantically weak relationship. It only states there is some relationship between the classes expressed without explicitly stating what kind of relationship. It could be contains, use, or inheritance. This is usually used in the analysis and design phases when some relationships among classes are still not clear or we just want to represent a general relationship among the classes.

2.1 New Definitions

In structured programming, one thinks in terms of input, function and outputs. In object-oriented programming (OOP), the approach is different -- a message is passed to an object requesting an operation on the object. Objects have methods and data fields, the methods specify the allowable operations on the object's private data, and the data fields specify the state information for the object. When a data field or methods change, it could affect other classes through message passing. We define an *affected class set* (ACS) to be the set of classes that could potentially be affected, *affected method set* of c (AMS[c]) to be the set of methods that could potentially be affected in class c, and *affect field set* of c (AFS[c]) as the set of data fields that could potentially be affected in class c.

If X is a data field or method, MREF(x) (*method reference set of x*) is the set of methods that reference x, in another words, method m references x as part of its implementation. MREF(x) represents the set of methods that could be affected by x, if x changes.

FREF(x) (*data field reference set of x*) is the set of data fields that are defined by the data field or method x. FREF(x) set are the set of data fields that can be potentially affected by x.

FDEF (*field definition set of x*) is the set of data fields that define a data field or method x.

The *affected method set* (AMS) of a class C is the set of all methods that reference any method in AMS or any field in AFS (*affected field set*) of C or any other class they use. The *affect field set* (AFS) of a class C contains all the fields that is defined or redefined by field in AFS or any methods in AMS of C. Induction is used to define these two sets.The *public affected method set* (PAMS) o f C is the AMS set that composed of public methods of C. The *public affected*

field set of C is the AFS set that composed of public data fields of C.

Suppose we want to see what impact a change could have on a system when the date members or methods in certain classes are proposed to be changed. First we initialize the ACS to the set of classes proposed to be changed, and initialize the AMS and AFS of each classe in the ACS. For example, the class C in ACS, its data member f_0 and method m_0 are proposed to change

$$AMS(C) = \{m_0\}$$

$$AFS(C) = \{f_0\}$$

Let's assume that at n -1 step, $AMS_{n-1}(C)$ contains all the affected methods in C, and $AFS_{n-1}(C)$ contains all the affected fields or data members in C

$AMS_{n-1}(C) = \{m | \forall m$, m is the affected method in class C$\}$

$AFS_{n-1}(C) = \{f | \forall f,$ f is the affected fields in class C$\}$

At step n,

AMS of C contains all the methods that reference any field in $AFS_{n-1}(c)$ plus any methods that reference any methods in $AFS_{n-1}(c)$, c is any class in the system

$$AMS_n(C) = \{m | \forall m \text{ in C}, \exists x, \exists c, \text{ s.t. m} \in MREF(x) \wedge x \in AFS_{n-1}(c)\}$$

$$\cup \{m | \forall m, \exists n, \exists c, \text{ s.t. } m \in MREF(n) \wedge n \in AMS_{n-1}(c)\}$$

AFS of C contains all the fields or data members that are defined by any field in $AFS_{n-1}(c)$ and any field that is defined by any methods in $AFS_{n-1}(c)$, c is any class in the system.

$$AFS_n(C) = \{f | \forall f \text{ in C}, \exists x, \exists c, \text{ s.t. f} \in FDEF(x) \wedge x \in AFS_{n-1}(c)\}$$

$$\cup \{f | \forall f \text{ in C}, \exists m, \exists c, \text{ s.t. } f \in FDEF(m) \wedge m \in AMS_{n-1}(c)\}$$

3. Algorithms

This section, we presentss the algorithms that analyze the ripple effect of the system when some component has been changed. We assume the existing system has been thoroughly tested. The algorithm calculate the transitive closure of ACS set of each class. It pick an unexamined class from the system, check all the classes that have direct relationship with this class according to encapsulation, inheritance characteristics, then add all the classes that could potentially be affected by this class in ACS set. The ACS of the entire system is the union of all the ACS sets of each class in the system. TotalEffect is the main algorithm that picks an unexamined class C from the system and calls other algorithm units. It calls FindEffectInClass(C) to calculate the AMS and AFS of C, FindEffectAmongClient(C) to determine the client classes that could be affected by C, FindEffectAmongChildren(C) the determine the subclasses that coulde be affected by C. All the affected classes are put into ACS set.

3.1 Total Effect

The TotalEffect Algorithm initializes the ACS set and AMS, AFS sets of each class in ACS using SetInit. SetInit also marks each class in the system as *dirty*, meaning they need to be checked by the algorithms. TotalEffect pick one dirty class from the system, mark them clean, meaning they have been checked, then uses FindEffectInClass(C) to analyze the effect within the class, uses FindEffectAmongChildren(C) to analyze the effect in the system according to inheritance, and uses the FindEffectAmongClient(C) to analyze the effect in the system according to encapsulation. We will explain FindEffectInClass(C), FindEffectAmongChildren(C) and FindEffectAmongClient(C) in detail in following sections. During algorithm execution, if the AMS or AFS sets of any clean class have increase, this clean is marked as dirty again for further examination. Figure 2, "Algorithm to calculate the total effect in the system" shows the high level flow of the algorithm.

3.2 Initialization

The Initialization algorithm will initialize the data structures according to the precondition of the algorithms. The user can specify what the methods and data fields of what classes they want to analyze. The SetInit will set the initial value of ACS to the classes of interest, the specified class AMS[C_i] to these affected methods of specified classes, and AFS[C_i] to the affected fields of specified classes. Figure 3, "Initialization algorithm" shows the SetInit algorithm.

3.3 Encapsulation

In traditional programming, the basic unit is a procedure. In object-oriented programming, methods or member functions are the actions that can be performed on objects. They manipulate and express the state of the object. They are

```
TotalEffect()
input: the set of changed classes and their changed methods and data fields.
output: the affected classes and their methods, data fields in the system.
/* conservative algorithms to find the ripple of the system */
BEGIN
     SetInit()
     FOR each class C in ACS
          IF C is not clean
               mark C clean.
               FindEffectInClass(C)
               FindEffectAmongChildren(C)
               FindEffectAmongClient(C)
          ENDIF
     ENDFOR
END TotalEffect
```

Figure 2: Algorithm to calculate the total effect in the system

```
SetInit()
BEGIN
     ACS = {the set of changed classes}
     Mark each class in the system dirty
     FOR each class in ACS  C_i
          AMS[ C_i ] = {the set of methods changed in  C_i }
          AFS[ C_i ] = {the set of fields changed in  C_i }
     ENDFOR
END SetInit
```

Figure 3: Initialization algorithm

the interface of a class to other classes and in many ways are not logically independent entities. Thus, we can treat classes as the basic unit for analysis, and focus on classes and objects. Encapsulation is a way to separate the implementation of a data object from its specification. An object does this by managing its own resources and limiting the visibility of what others should know. An object publishes a public interface that defines how other objects or applications can interact with it. An object also has a private component that implements the methods. The object's implementation is encapsulated -- that is, hidden from the public view [OH96]. In the presence of encapsulation, the only way to observe the state of an object is through its interface (public methods). The class hides the properties of its instances to conceal the data structure and the details of implementation. All the features of an object are usually hidden, such that the only way the state can be examined or modified is by invoking its interface formed by its public properties. The interface is a basis for a protocol that objects use to communicate with each other by requesting an object to invoke one of its operations. Methods and data members in the class can see all the properties within the class.

For each class C, AMS[C] contains all the methods that could be affected by specified changes. AFS[C] holds all the data fields that could be affected by specified changes. Since the only way to observe the state of an obejct or operate on an object is through its public methods or data fields, this object's clients can only be directly affected by the changes in the public methods or data fields. PAMS[C] contains all the public methods that could be affected. PAFS[C] holds all the public fields that could be affected. obviously, $PAMS[c] \subseteq AMS[c]$, $PAFS[c] \subseteq AFS[c]$.

Operations in objects interact with each other by modifying the state of their objects. The control flow analysis or data analysis techniques are not directly applicable to the object level, since there is no sequential order in which the operation will be invoked.

Finding Effect Within a Class (FindEffectInClass)

When a method or data field in class C changes, the effects within the class C can be found by FindEffectInClass(C). Since the execution within each method is still sequential, we can apply CFG and DFG techniques to find the MREF, and FREF sets of C. FindEffectInClass checks each method m in class C that is not in *affected methods set* (AMS), and each data field f that is not in *affected field set* (AFS). If m references any methods in AMS ($MREF(m) \cap AMS(C) \neq \text{Empty}$) or m references any data fields in AFS ($FREF(m) \cap AFS(c) \neq \text{Empty}$), m could be affected by the changes in AMS and AFS. So it will be added to AMS and to PAMS if m is public. If data field f references any data field in AFS ($FREF(f) \cap AFS(c) \neq \text{Empty}$) or references any method in AMS ($MREF(f) \cap AMS(c) \neq Empty$), add f to AFS and to PAFS if f is public. This sounds reasonable, but unfortunately this algorithm has a flaw. Assume a class has methods m1, m2, m3, m4, m5. m1 and m2 are in AMS set; if m3 references m5 and m5 references m2, m3 references m2 indirectly, $m2 \in AMS$, so m3 should belongs to AMS. But when we check m3, since m5 has not been checked yet, m3 could not find any reference in AMS set, so the algorithm thinks it is clean and fails to put it in AMS. To fix this problem, we put the methods or data fields that cannot find any reference in AMS or AFS set in a temporary clean set. After having checked all the methods and data fields in the class, the algorithm examines all the methods and data fields in this clean set, to check whether there are more methods or data

fields that could be affected. See Figure 4, “Algorithms to calculate change effects inside class” for detail information

```
FindEffectInClass(C)
/* Effect in class will find the effect within the
   class, if certain data members or methods have changed */
input: the AMS and AFS sets in C before this algorithm is executed.
      They could come from initialization or the execution result from previous
      iteration.
output: the AMS and AFS sets in Class C after the execution of this algorithm.
      They include the original members plus any newly added members
BEGIN
     Analyze the CFG and DFG of each methods, construct MREF & FREF sets for each
      methods and data fields.
     /* initial searching of methods and data fields */
     FOR each method m in C
     BEGIN
          IF (m ∈ AMS) and ((MREF(m) ∩ AMS(C) ≠ Empty) or
              (FREF(m) ∩ AFS(c) ≠ Empty))
               AMS(c) = AMS(c) ∪ {m}
               if m is public member
                         PAMS(c) = PAMS(c) ∪ {m}
          ELSE CleanMethod = CleanMethod ∪ {m}
          ENDIF
     ENDFOR
     FOR each field f in C
     BEGIN
          IF (f ∉ AFS) and ((FREF(f) ∩ AFS(c) ≠ Empty) or
              (MREF(f) ∩ AMS(c) ≠ Empty))
          BEGIN
               AFS(c) = AFS(c) ∪ {f}
               IF f is public attribute
                     PAFS(c) = PAFS(c) ∪ {f}
          END ELSE   CleanField = CleanField ∪ {f}
          ENDIF
     ENDFOR
     FOR each method m in CleanMethod
     BEGIN
          IF ( MREF(m) ∩ AMS(C) ≠ Empty ) or ( FREF(m) ∩ AFS(c) ≠ Empty )
          BEGIN
               AMS(c) = AMS(c) ∪ {m}
               IF m is public member
                    PAMS(c) = PAMS(c) ∪ {m}
          ENDIF
     ENDFOR
     FOR each field f in CleanField
     BEGIN
          IF (FREF(f) ∩ AFS(c) ≠ Empty) or (MREF(f) ∩ AMS(c) ≠ Empty)
          BEGIN
               AFS(c) = AFS(c) ∪ {f}
               IF f is public attribute
                    PAFS(c) = PAFS(c) ∪ {f}
          ENDIF
     ENDFOR
END FindEffectInClass
```

Figure 4: Algorithms to calculate change effects inside class

Finding Effects Among Clients (FindEffectAmongClients)

If class A sends messages to class B, A *uses* B. We can say class A is class B's *client*. Encapsulation builds a wall between the class and its clients. Assume the current affected class is C_0 and we want to determine which classes that use C_0 will be affected. Because of the encapsulation, the clients of C_0 can only access this class through its public members, which means its clients can only be affected by this class's PAMS and PAFS sets.

FindEffectAmongClients examines each client class of C_0 and puts any methods or data fields that reference methods or data fields in PAMS or PAFS of C_0 into their own AMS or AFS set. If any of these methods or data fields are public, they are put into PAMS and PAFS of these client classes. FindEffectAmongClients finds other methods and data fields that might be affected by the newly added affected methods and data members in AMS and AFS in each client class by calling their FindEffectInClass().

We have marked each class as *dirty* at the initialization. Classes that are dirty need to be checked by the algorithms as an initial class (not checked as a client or child of the class being analyzed). We define OLDAMS and OLDAFS as the two sets that contains the AMS and AFS before FindEffectAmongClient start its process. At the end, the algorithm checks whether there are any new methods or data fields that have been added to the AMS and AFS set by compare the AMS and AFS with the OLDAMS and OLDAFS. If there are new methods or data fields in a client class, it means this client class might influence more classes in the system by these newly added members. This class needs to be checked again by the algorithms, so it is marked as dirty to be picked by the main loop in TotalEffect(). See Figure 5, "Algorithm to calculate the change effect among clients" for details of this algorithm.

3.4 Inheritance

Inheritance is the mechanism that allows the developer to create new child classes -- known as subclasses or derived classes -- from existing parent classes. Inheritance represents a hierarchy of abstractions, in which a subclass inherits from one or more super classes. The child class shares the structure or behavior defined in its parent class. The child class can express differences with its parent class by modifying and adding properties.

Different languages accept different inheritance schemes (strict inheritance, subtyping, subclassing etc.). *Strict inheritance* is the simplest inheritance scheme; it keeps the exact behavior of its parent. The inherited properties cannot be modified, the derived class can only be redefined by adding new properties. *Subtyping* is the most commonly used scheme. In addition to properties of strict inheritance, subtyping allows for the redefinition of the inherited properties when the parent's operation is not appropriate for the subclass. Subclassing is a scheme of inheritance in which the derived class is not considered as a specialization of the base class, but as a completely new abstraction that bases part of its behavior on a part of another class. This scheme is also called *implementation inheritance*. The derived class can

```
FindEffectAmongClients (C0)
input: the ACS set and AMS, AFS sets in C before this algorithm is executed.
They could come from initialization or the execution result from previous iteration.
output: The expanded ACS, and the expanded AMS, AFS, PAMS, PAFS sets of the classes
      that belongs to ACS.
FOR each class C that uses C0
BEGIN
     OLDAMS[C] = AMS[C]
     OLDAFS[C] = AFS[C]
     FOR each methods m in C
     BEGIN
          IF ( MREF(m) ∩ PAMS(C0) ≠ Empty) or ( FREF(m) ∩ PAFS(C0) ≠ Empty )
          BEGIN
               AMS(C) = AMS(C) ∪ {m}
               IF m is public member
                    PAMS(C) = PAMS(C) ∪ {m}
          ENDIF
     ENDFOR
     FOR each field f in C
      begin
          if ( FREF(f) ∩ PAFS(c) ≠ Empty ) or ( MREF(f) ∩ PAMS(c) ≠ Empty )
          begin
            AFS(c) = AFS(c) ∪ {f}
            if f is public attribute
              PAFS(c) = PAFS(c) ∪ {f}
          endif
      endfor
      FindEffectInClass(C)
      if ( (OLDAMS[C] ≠ AMS[C]) or (OLDAFS[C] ≠ AFS[C]) )
      begin
        ACS = ACS ∪ {C}
       mark C dirty
      endif
     end FindEffectAmontClients
```

Figure 5: Algorithm to calculate the change effects among clients

therefore choose not to inherit all the properties of its parent. In this paper, we assume the language is using subtyping. For other inheritance schemes, the algorithm described in this section needs minor adjustment.

Inheritance can be thought of as an incremental modification technique that combines a parent P with a modifier M to get a resulting class R. $R = P \oplus M$. [WZ88]

The subclass designer specifies the modifier, which may contain various types of attributes that alter the parent class to get the resulting subclass. Although modifier M transforms a parent class P into a resulting class R, M does not totally constrain R. We must also consider the inheritance relation since it determines the effects of composing the attributes of P and M and mapping them into R. The inheritance relation determines the visibility, availability and format of P's attributes in R. Since inheritance is deterministic, rules can be constructed to identify the availability and visibility of each attribute.

When a subclass redefine its parent's method, it can either totally overwrite the service of its parent's method or expand its parent's service. The impact of the parent's method on this subclass will be different if the subclass expands the parent's method in different ways. If the subclass totally reimplements its parent's service without using its parent's service, the change in the parent's method will not affect the subclass. If the subclass expands its parent's service based on the service the parent's method provides, any changes in the parent's method could affect this subclass. Because of this, we extend Harrold and McGregor's attributes classification even further by splitting the redefine and virtual redefine into extended redefine, total redefine, virtual extended redefine, and virtual total redefine. As a result, methods in subclasses are divided into following categories to gain more control: New attribute, Inherited attribute, Extended-redefined attribute, Total-redefined attribute, Virtual-new attribute, Virtual-inherited attribute, Virtual-extended-redefined attribute, virtual total-redefine attribute. These categories are defined below.

New attribute: A is an attribute that is defined in M but not in P or A is a member function attribute in M and P but with different signature. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Inherited attribute: A is defined in P but not in M. In this case, A is bound to the locally defined attribute in P. A is accessible within R and accessible outside R if A is public; A is accessible both within and outside P.

Extended redefined attribute: A is defined in both P and M with the same signature. The A in M will extend the functionality of A in P by using the services of A in P. In this case, A is bound to the locally defined attribute in M. A is accessible inside R and accessible outside R if A is public; A is not accessible in P.

Total redefined attribute: A is defined in both P and M where A's signature is the same in M and P. The A in M will replace the functionality of A in P by implementing the services without using the A in P. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Virtual new attribute: A is specified in M but its implementation may be incomplete in M to allow later definitions or A is specified in M and P and its implementation may be incomplete in P, but A's signature differs in M and P. In this case, A is bound to the locally defined attribute in M. A is accessible within R and accessible outside R if A is public; A is not accessible in P.

Virtual inherited attribute: A is specified in P but its implementation may be incomplete in P to allow later definition, and A is not defined in M. In this case, A is bound to the locally defined attribute in P. A is accessible within R and accessible outside R if A is public; A is accessible both inside and outside P.

Virtual extended redefined attribute: A is specified in P but its implementation may be incomplete in P to allow for later definition and A is defined in M with the same signature as A in P. The A in M will extend the functionality of A in P by using the services of A in P in M's implementation. In this case, A is bound to the locally defined attribute in M. A is accessible inside and outside R if A is public; A is not accessible in P.

Virtual total redefined attribute: A is specified in P but its implementation may be incomplete in P to allow for later definition and A is defined in M with the same signature as in P. The A in M will replace the functionality of A in P by implementing the services without using the A in P. In this case, A is bound to the locally defined attribute in M. A is accessible inside and outside R if A is public; A is not accessible in P.

The inheritance relation determines visibility, availability and format of P's attributes in R. A language may support more than one inheritance mapping by allowing specification of a parameter value to determine which mapping is used for a particular definition.

Polymorphism is the possibility of a reference to denote instances of various classes. It is usually constrained by inheritance. Polymorphism means that the same method can do different things, depending on the class that implements it. It lets two similar objects be viewed through a common interface and allows subclasses to override an inherited method without affecting the ancestor's methods [OH96]. If the inheritance scheme is subtyping, the denoted objects all have at least the properties of the root class of the hierarchy. Thus an object belonging to a derived class could be substituted into any context in which an instance of the base class appears, without causing a type error in any subsequent execution of the code. Martin [Mar95] calls this total polymorphism. It is described by the Liskov Substitution principle: If for each object o1 of type S, there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T. Less formally, the software can always pass a pointer or reference to a derived class to a function that expects a pointer or reference to a parent class. Since polymorphic names can denote object of different classes, it is impossible to predict which class will be executed until run time. McGregor [MC94] calls it strict inheritance and describes it from an another point of view. Strict inheritance means:

> Pre-conditions on a particular method in a class must be no stronger than those of the same method in a parent class.
>
> Post-condition on a particular method in a class must be no weaker than those on the same method in a parent class.
>
> The invariant for a class must be a superset of the invariant for a parent's class.

Now we can analyze how the changes in the ACS propagate through parent and children classes by inheritance and polymorphism. From the attribute categories above, we know any changes in a child will not affect its parent because its parent cannot access the methods or data fields of its children. However, changes in a parent can affect its children. Smith and Roberson [SR90] think a change to a parent class can potentially affect all descendants. Let us analyze it through the inheritance categories of the methods in subclasses.

- If the method or data field A in a child class is a new attribute, A is defined in M but not in P or the signature of A in M and P is different. Since A is not accessible in P, the new attribute in R will not affect the A in P. But it will affect P's children.

- If the method or data field A in a child class is an inheritance attribute, A is locally bound to P. In this situation, If A in P changes, R needs to retest it, because the context of A in P is different than the context of A in R.

- If the method or data field A in a child class is a total redefined attribute, M redefines A without using P's version of A. So A's change in P will not affect A in R. But if A in R uses other methods in AMS[P], it will still be affected.

- If the method or data field A in a child class is an extended redefined attribute, M extended the functionality of A in P by adding extra functionality to A. The A in P is invoked in M. So any change of A in P will affect the A in R. A's change in R will not affect its parent P since its parent either does not have A or its version of A has a different signature. A's change in P will affect R, so R is in ACS.

Figure 6, "Algorithm to calculate the change effect among subclasses" is the algorithm that finds the effect through inheritance and polymorphism:

3.5 Complexity Estimation

Assuming the number of classes in the system is m. Let

$nm = max($*number* of methods in Class i) i = 1 .. m formula 1

$nf = max($*number* of data fields in Class i) i = 1 .. m formula 2

$n = max(nm, nf)$ formula 3

By analyzing Figure 2, "Algorithm to calculate the total effect in the system", we can tell that the overall complexity

```
FindEffectAmongChildren(C_p)
BEGIN
    FOR each class C_c that inherited from C_p
        FOR each method m in C_c {
            case ( inheritance type of m )
            Extended Redefine:
                IF ( m_p ∈ C_p ∩ m_p ∈ AMS[C_p] )
                    AMS(C_c) = AMS(C_c) ∪ {m}
                    IF m is public
                        PAMS(C_c) = PAMS(C_c) ∪ {m}
                    ENDIF
            Total Redefine:
            Virtual Extended Redefine:
                IF ( m_p ∈ C_p ∩ m_p ∈ AMS[C_p] )
                    AMS(C_c) = AMS(C_c) ∪ {m}
                    IF m is public
                        PAMS(C_c) = PAMS(C_c) ∪ {m}
                    ENDIF
            Virtual Total Redefine:
                IF ( m_p ∈ C_p ∩ m_p ∈ AMS[C_p] )
                    AMS(C_c) = AMS(C_c) ∪ {m}
                    IF m is public
                        PAMS(C_c) = PAMS(C_c) ∪ {m}
                    ENDIF
            Inherit, Virtual inherit:
                IF (m_p ∈ C_p ∩ m_p ∈ AMS[C_p] )
                    AMS(C_c) = AMS(C_c) ∪ {m}
                    IF m is public
                        PAMS(C_c) = PAMS(C_c) ∪ {m}
                    ENDIF
            OTHERS:
            /* all other cases will not be affected by the change of m in A */
            ENDCASE
            IF (m ∉ AMS[C_c] ) and (( FREF(m) ∩ PAFS(c) ≠ Empty ) or
                ( MREF(m) ∩ PAMS(c) ≠ Empty ))
                AFS(c) = AFS(c) ∪ {m}
                IF m is public attribute
                    PAMS(c) = PAMS(c) ∪ {m}
            ENDIF
        ENDFOR /* enf of for each method */
    ENDFOR /* end of for each class */
END FindEffectAmongChldren
```

Figure 6: Algorithm to calculate the change effects among subclasses

of the whole algorithm is:

$O(TotalEffect) = max(O(SetInit)$, O(m)*O(the body of for loop in Figure 2)) Formula 4 .

Since SetInit has to mark each class in the system as *dirty*, the worst case complexity of SetInit is O(m). The complexity of the body of *for loop* in Figure 2 is equal to max(O(FindEffectInClass), O(FindEffectAmongChildren), O(FindEffectAmongClients)). By analyzing the Figure 4, "Algorithms to calculate change effects inside class", we get the worst case complexity of FindEffectInClass(C) is $O(m)$. By analyzing Figure 5 and Figure 6, we get the worst case complexity of FindEffectAmongChildren is $O(mn)$ and the worst case complexity of FindEffectAmongClients is $O(m^2 n)$. So O (for loop in Fiture) = $O(m^2 n)$. From Formula 4, we get $O(TotolEffect) = O(m^3 n)$.

The overall algorithms is actually calculate the transitive closure of all the affected classes. As we mentioned before, some classes have been checked could be remarked as *dirty*, if their AMS or AFS expanded. Since the number of members in AMS or AFS cannot be greater then *n*, so $O(TotolEffect) = O(m^3 n) \times O(n) = O(m^3 n^2)$.

4. Algorithm Improvement

The algorithms section 3 offered is a conservative approach to estimate the system-wide impacts of proposed changes. Because certain types of changes will not necessarily affect other parts of the system, some classes that are putted into ACS by section 3 may not necessarily affect other classes. In follwoing section, we are going to categorize changes that can be applied to object-oriented software, analyze the characteristics of these categories, and discuss in detail what kinds of changes will affect other parts of the system and what kinds of changes will not affect other parts of the system in following section.

There are many different kinds of changes that can be applied to object-oriented software. We categorize these changes by specifying the types of changes that could be applied to data members, methods, classes and objects. Each of these different types of changes is assigned one of the following six attributes according to their influences on other classes in the system:

- Contaminate_all: This type of change will affect the data members and methods in any classes that are related to the current changed class. These classes could be client, subclass of current changed class or be the changed class itself.
- Contaminate_current: This type of change will only affect the data members and methods in the current class that contains the change.
- Contaminate_children: This type of change will only affect subclasses that are derived from the changed class.
- Contaminate_client: This type of change will only affect client classes that use the changed class.
- Contaminate_none: This type of change will not affect any data members or methods belonging to either client classes or subclasses of the changed class, or belonging to the changed class itself.

These attributes can be represented by an attribute byte, in which Contaminate_current, Contaminate_children, Contaminate_client each occupies one bit. If a change will affect all related classes (Contaminate_all), its attribute byte is 0x07 (Contaminate_current | Contaminate_children | Contaminate_client). If a change affect client and child classes, its attribute type is 0x06 (Contaminate_children | Contaminate_client = 0x04 | 0x 02 = 0x06). If a change does not affects any other class, its attribute is 0x00.

Figure 7, "Change Category" summaries all the changes categories and the relationships among them. We will explain each of these category in detail in the following section.

4.1 The Type of Changes That Could Be Applied to Data Members

We classify the potential changes based on the syntactic element changed and the action used to make the change. The characteristics of each type of change and how each type can influence other parts of hte system is analyzed. Although we have tried to cover all cases, we have no basis on which to claim this listing is exhaustive.

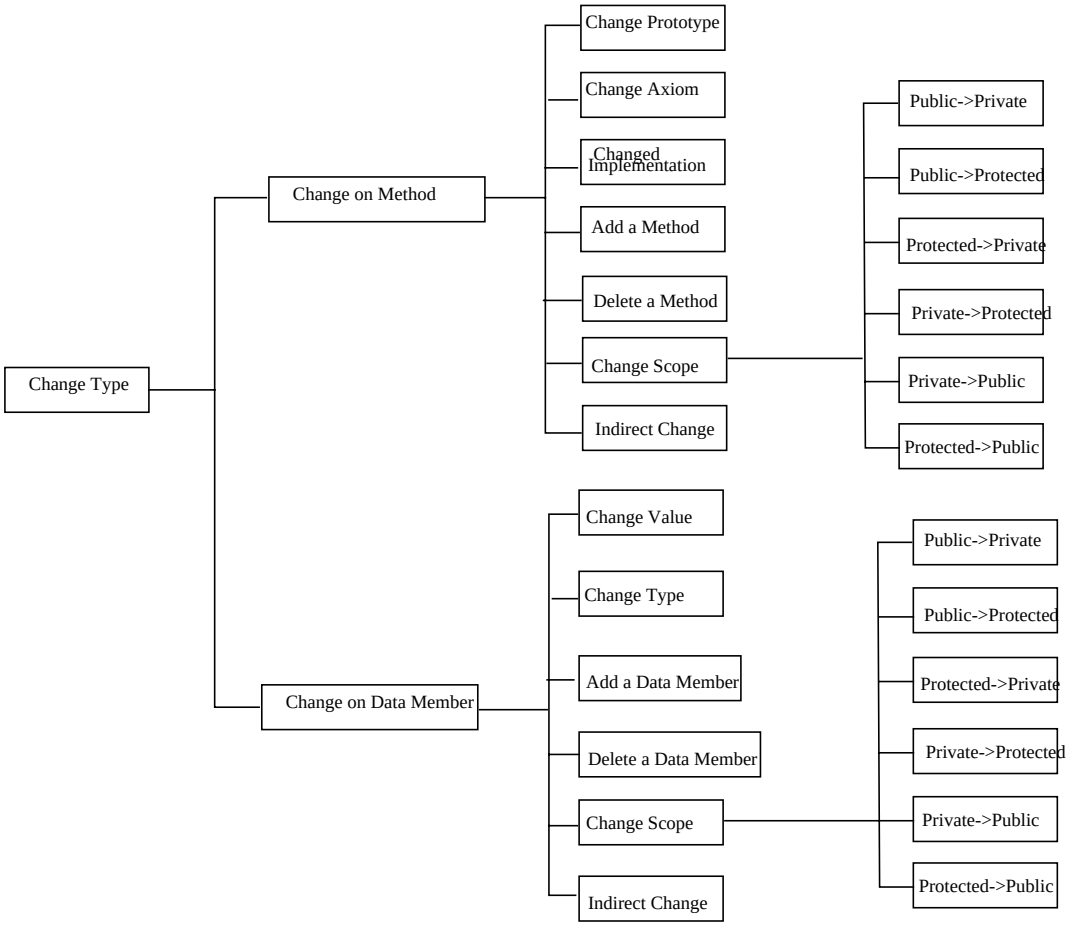


Figure 7: Change Category

1. **Value changed**: If the value of the data member is changed, it will change the state of the object. This kind of change may or may not affect other data members or methods, depending on whether this change will change the state of the object. If the state of the object is changed, the execution of the program could lead to some paths that it have never been executed. So if the change causes the object to change its state, it will contaminate all of the related methods in the changed object (Contaminate_current = 0x01). Otherwise, if it does not cause the object to change its state, it will not contaminate any of the related classes (Contaminate_none = 0x00).
2. **Type changed**: If the type of the data member is changed, it will affect the methods or other data members that reference it. If this data member is public, it will affect the data members or methods from the changed class, its clients, and its children (contaminate_all = 0x07). If it is protected, it will affect the data members or methods from the changed class and its children (contaminate_children | contaminate_current = 0x03). If it is private, it will affect the data members or methods from the changed class only (contaminate_current = 0x01).
3. **Scope changed:** Let us assume the language has three levels of scope: public, protected and private. Public data members or methods constitute the interface of the class and can be seen by other classes. Protected data members or methods can only be seen by data members or methods within the class or from this class' derived class. Private data members or methods can only be seen by the data members or methods within the class. There are several possible scope changes:

I) Public --> Private

II) Public --> Protected

III) Protected --> Private

IV) Protected --> Public

V) Private --> Public

VI) Private --> Protected

I) Changing a data member from Public to Private will affect any client classes and subclasses that reference this data member, because it will not be available after it disappears from public section. Since the data members and methods in the changed class can still see this data member, they will not be affected by this change. The attribute is 0x06 (contaminate_client | contaminate_children = 0x06).

II) Changing a data member from Public to Protected will affect any client classes that reference this data member, but not the data members and methods in subclasses and in the changed class. Because this data member will not be available for any data members and methods in client classes but still available for those in subclasses and changed class, this change will only affect the client classes of the changed class. The attribute of this change is contaminate_client (0x04).

III) Changing a data member from Protected to Private will affect all the subclasses that are derived from this class. Changing the data member from protected to private section make this data member not available for any data members and methods in its subclasses. The attribute of this is Contaminate_children (0x02).

IV) V) VI) will not affect any other classes except to reveal the state of the object. So its attribute is contaminate_none (0x00).

4. **Delete a data member:** When a data member is deleted, it will not be available to its client anymore, so all its clients will be affected. If this date member is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all = 0x07). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children | contaminate_current = 0x03). If it is private, it will only affect the data members or methods from the changed class (contaminate_current = 0x01).
5. **Add a data member:** When a data member is newly added, it does not have any classes to use it yet. So we assume it is non-contaminative to its client, but , according to Liskov principle, its children have to know the new data member if it is public. Its attribute is contaminate_children (0x02).
6. **Affected by other data members or methods**: If a data member references other data members or methods that are contaminative, this data may be affected. It is very hard to predict what kind of changes the change of the referenced data member or methods can bring to this data member. The change may be able to fixed by programmer locally, such that this data member will not affect other parts of the system. In that sense, it is not contaminative. For example, when a referenced data member has been changed from public to private, that data member will not be available for its clients to reference anymore. The developer can replace this data member by a corresponding member function that retrieves the value of this data member. So its clients can just substitute the data member with the corresponding method, and keep their interfaces the same. In this situation, its clients will not propagate this change further. But it may need more dramatic changes that cause its clients to have to change their interfaces. So we assume that if this data member is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all = 0x07). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children | contaminate_current = 0x03). If it is private, it will affect the data members or methods from the changed class only (contaminate_current = 0x01).

The Type of Changes That Could Be Applied to Method

The types of changes that could happen to method are:

1. **Signature changed:** If the signature of a method has changed, for example, input, output parameters have been added or deleted, it will affect any methods or data members that relate to this method. If the method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed

class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class (contaminate_current).

2. **Axiom changed:** If the preconditions, postConditions, or axioms are changed, this will change the behavior or semantics of the method. This may or may not affect the methods or data members that reference this method. If the method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will affect the data members or methods from the changed class only (contaminate_current).
3. **Implementation changed:** This type of change affects the details of the implementation but not the interface. The semantics and behavior may or may not be changed. If this method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class (contaminate_current).
4. **Delete a method:** When a method is deleted, it is no longer available to its clients, so all of its clients are affected. If the method is public, it will affect the data members or methods from the changed class, its clients, and its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class(contaminate_current).
5. **Add a method:** When a method is newly added, it does not have any classes that use it yet. So we assume it is non-contaminative to its client, but its children have to know the new method. Its attribute should be contaminate_children.
6. **Scope changed:** Here are the possible scope changes relevant to a method

   I) Public --> Private

   II) Public --> Protected

   III) Protected --> Private

   IV) Protected --> Public

   V) Private --> Public

   VI) Private --> Protected

   I) Changing a method from Public to Private will affect any client classes and subclasses that reference this method, because this method will not be available for them after they disappear from public section. The attribute is 0x06 (contaminate_client | contaminate_children = 0x06).

   II) Changing a method from Public to Protected will affect any client classes that reference this method, but not the data members and methods in subclasses and in the changed class. Because this method will not be available for any client classes but still available for subclasses and the changed class. The attribute of this change is Contaminate_client (0x04).

   III) Changing a method from Protected to Private will affect all the subclasses that derived from this class. Changing the method from protected to private section make this data member not available to its subclasses. The attribute of this is Contaminate_children (0x02).

   IV) V) VI) will not affect any other classes except reveal the state of the object. So its attribute is contaminate_none (0x00).
7. **Affected by other data member or methods:** If a method is affected by other data members or methods, the state of the object will change, and this method will become contaminative. If it is affected by other methods, the referenced method's changed behavior can change the behavior of this method. If this method is public, it will affect the data members or methods from the changed class, from its clients, and from its children (contaminate_all). If it is protected, it will affect the data members or methods from the changed class and from its children (contaminate_children, contaminate_current). If it is private, it will only affect the data members or methods from the changed class(contaminate_current).

Other Changes

1. **Add a class:** When a class is newly added, it does not yet have any classes that use it. So we assume its attribute is Contaminate_none.
2. **Delete a class:** When a class is deleted, all the data members and methods will not be available to any related classes any more, so all of the classes related to it will be affected. The attributes of all the data members and methods is contaminate_all.

4.2 Use Cases

This section illustrates these algorithms through an example. The classes are heavily interrelated with each other to show how the algorithms work. Assume there are four classes in the system A, B, C, and D. The methods and data members and pieces of implementation pseudo-code of each classes are shown on the Figure 9, "Class Diagram of the Sample System". we analyze the impact to the system of changing the type of fa1 in A.

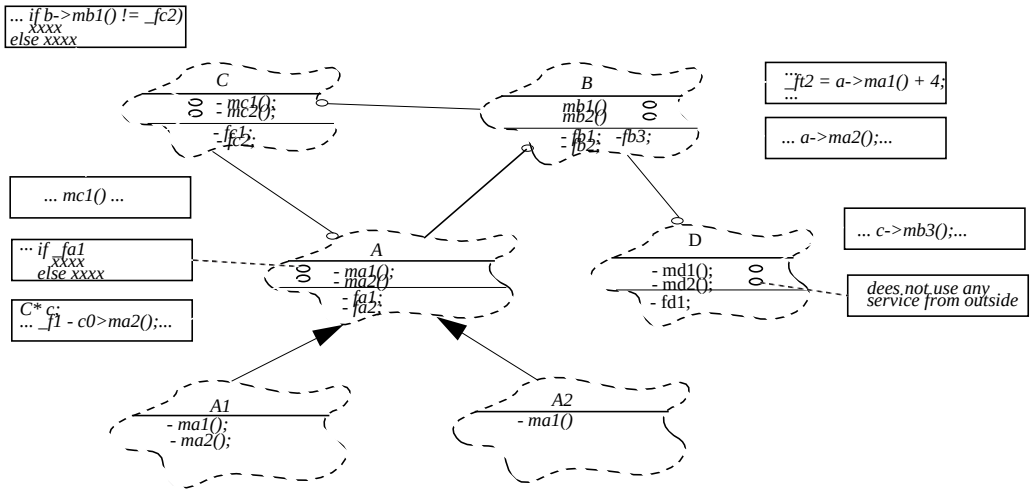


Figure 8: Class Diagram of the Sample System

3. Set the initialize value of different sets, put A in ACS set, mark dirty, and initialize the AMS and AFS of A

   $AMS[A] = \{\ \}$

   $AFS[A] = \{\ \_fa1\}$

   $ACS = ACS \cup \{A(Dirty)\}$

4. Pick A from ACS and mark it clean

   a) Check the effect in side the class. we get:

   $AMS[A] = \{\ ma1\}$

   $AFS[A] = \{\ \_fa1\}$

   $ACS = \{A(Clean)\}$

   b) Check the subclass of A: A1

   $AMS[A1] = \{\ \}$

   $AFS[A1] = \{\ \}$

   $ACS = \{A(Clean)\}$

   c) Check the subclass of A: A2

   $AMS[A2] = \{\ ma1\}$

   $AFS[A2] = \{\ \}$

   $ACS = \{A(Clean), A2(Dirty)\}$

d) Check the client of A: B

$AMS[B]$ = { mb1}

$AFS[B]$ = { _fb1}

$ACS = \{A(Clean), A2(Dirty), B(Dirty)\}$

5. Pick one dirty class A2 from ACS, mark it clean

a) Check inside class A2

$AMS[A2]$ = { ma1}

$AFS[A2]$ = { }

$ACS = \{A(Clean), A2(Clean), B(Dirty)\}$

6. Pick one dirty class B from ACS, mark it clean

a) Check inside class B

$OLDAMS[B]$ = { mb1}

$OLDAFS[B]$ = { _fb1}

$AMS[B]$ = { mb1}

$AFS[B]$ = { _fb1}

$ACS = \{A(Clean), A2(Clean), B(Clean)\}$

b) Check the client class of B: C

$AMS[C]$ = { mc1, mc2}

$AFS[C]$ = { }

$ACS = \{A(Clean), A2(Clean), B(Clean), C(Dirty)\}$

c) Check the client of B: D

$AMS[D]$ = { }

$AFS[D]$ = { }

$ACS = \{A(Clean), A2(Clean), B(Clean), C(Dirty)\}$

7. Pick one dirty class C from ACS, mark it clean

a) Check inside class C

$AMS[C]$ = { mc1, mc2}

$AFS[C]$ = { }

$ACS = \{A(Clean), A2(Clean), B(Clean), C(Clean)\}$

b) Check the client of C: A

$OLDAMS[A]$ = { }

$OLDAFS[A]$ = { _fa1}

$AMS[A]$ = { ma1, ma2}

$AFS[A]$ = { }

Since OLDAMS[A] != AMS[A]

$ACS = \{A(Dirty), A2(Clean), B(Clean), C(Clean)\}$ // mark A dirty again

8. Pick one dirty class A from ACS, mark it clean

a) Check inside A -- No change in A's AMS and AFS set

b) Check the subclass of A: A1 -- No change in A1's AMS and AFS set

c) Check the subclass of A: A2

$OLDAMS[A2]$ = { ma1}

$OLDAFS[A2]$ = { }

$AMS[A2]$ = { ma1, ma2}

$AFS[A2]$ = { }

Since OLDAMS[A2] != AMS[A2]

$ACS = \{A(Clean), A2(Dirty), B(Clean), C(Clean)\}$ // mark A2 dirty again

d) Check the client class of A: B

$OLDAMS[B] = \{ mb1 \}$

$OLDAFS[B] = \{ \}$

$AMS[B] = \{ mb1, mb2 \}$

$AFS[B] = \{ \_fb1 \}$

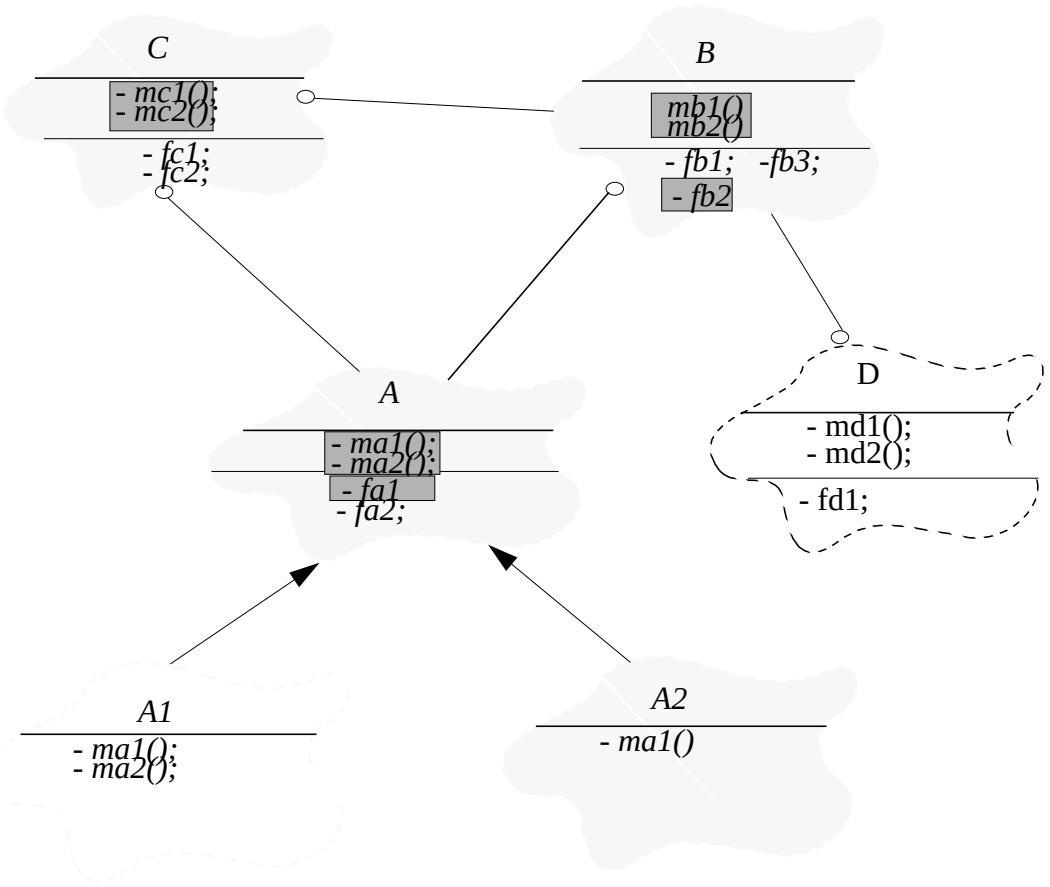
Since OLDAMS[B] != AMS[B] and OLDAFS[B] != AFS[B]

$ACS = \{A(Clean), A2(Dirty), B(Dirty), C(Clean)\}$ // mark B dirty again

9. Pick one dirty class A2. mark it clean
   a) Check inside A -- No change in A2's AMS and AFS set

      $ACS = \{A(Clean), A2(Clean), B(Dirty), C(Clean)\}$
10. Pick one dirty class B, mark it clean
    a) Check inside B -- No change in B's AMS and AFS set
    b) Check client class of B: C and D -- No change in C and D's AMS and AFS set

    $ACS = \{A(Clean), A2(Clean), B(Clean), C(Clean)\}$
11. There is no dirty class in ACS set, the algorithm display the affected class and the affected data members and methods associated with them (See Figure 10, "Result of the algorithm")



* The affected classes liked A, A2, B, C are shaded.
* The affected data memeber or methods of affected class are shaded in different gray level.

Figure 9: Result of the algorithm

5. Conclusion and Future Work

In this paper, we have analyzed the characteristics of the object-oriented software to understnad how encapsulation, inheritance, and polymorphism influence the change propagation. We categorize the different kinds of changes that could be applied to object-oriented software, and assign each type of change an influence attribute according to how this type of change influences other objects in the system. A simple and conservative approach was first described, then ways to optimize the algorithms according to the change type, the change attributes associated with it. The complete set of algorithms that calculate the change propagation within the class, between the client and server class, and between the parent and children class are described in a technical report [xx].

The technique described in this paper can be used by software developers to run “what if” analysis on different “update” proposals for the software system, and choose the one that is most cost effective. It can also be used by software testers to find what areas are affected by the changes,, so they can test only the affected areas and still feel confident about the quality of the software.

The algorithms described in this paper are detailed enough to implement them in the real environment. In the future,

we hope to develop a metric system to measure the impact of the proposed changes quantitatively, and develop an analysis tool that implements these algorithms. Another interesting plan is to expand this technique to a distributed object environment, and analyze how changes propagate across heterogenous networks, databases, operating systems, and languages.

6. References

[Boo94] Grady Booch, “OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS,” SECOND EDITION, Benjamin/Cummings Publishing Company 1994

[Rum91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[Bar94] Stephane Barbey, Alfred Strohmeier, “The Problematic of testing Object-Oriented Software”, SQM’94 Second Conference on Software Quality Management, volume 2, Edinburg, Scotland, UK, 1994

[Bar+94] Stephane Barbey, Alfred Strohmeier, “Open Issues in Testing Object-Oriented Software”, ECSQ’94, Basel, Switzerland, October 1994

[TR92] C. D. Turner and D.J. Robson, “The Testing of Object-Oriented Programs”, Technical Report: TR-13/92, Computer Science Division School of Engineering and Computer Science (SECS) University of Durham, England

[Mar95] Robert C. Martin, “Designing Object-Oriented C++ Applications Using The Booch Method”, Prentice Hall, Inc. 1995

[Ofut95] Alisa Irvine and A. Jefferson Offutt, “The Effectiveness of Category-Partition Testing of Object-Oriented Software”, ISSE Department George Mason University, Fairfax, VA 22030.

[Fie89] S.P. Fiedler. Object-oriented unit testing. Hawlett-Packard Journal, 40:69-74, April 1989.

[Per90] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. Journal of OOP, Jan/Feb 1990.

[Wey88] E. Weyuker. The evaluation of program-based software test data adequacy criteria. Communications of the ACM, June 1988.

[CM90] T. E. Cheatham and L. Mellinger. Testing object-oriented software system. In 1990 ACM Eighteenth Annual Computer Science Conference, pages 161-165, February 1990.

[WZ88] P. Wegner and S. B. Zdonik, “Inheritance as an incremental modification mechanism or what like is and isn't like,” Proceedings of ECOOP'ii, pp.55-77, Springer-Verlag, 1988.

[SR90] M. D. Smith and J. J. Robson. Object-Oriented programs - the problems of validation. In Proceedings of the 1990 IEEE Conference on Software Maintenance pages 272-281, San Deiego, CA, Nov 1990.

[HM] Mary Jean Harrold and John D. McGregor, “Incremental Testing of Object-Oriented Class Structures”, Clemson University.

[KL90] Bogdan Korel, Janusz Laski, “Dynamic Slicing of Computer Programs”, J. Systems Software 1990.

[SB94] Stephane Barbey, Alfred Strohmeier, “The Problematic of Testing Object-Oriented Software”, Swiss Federal Institute of Technology, Computer Science Department, Software Engineering Laboratory, EPFL-DI-LGL, 1015 Lausanne, Switzerland.

[SB94] Stephane Barbey, Mauel M. Ammann and Afred Strohmeier, “Open Issues in Testing Object-Oriented Software”, Swiss Federal Institute of Technology, Computer Science Department, Software Engineering Laboratory, Published in ECSQ’94 October 1994.

[OH96] Robert Orfali, Dan Harkey, Jeri Edwards, “The Essential Distributed Objects Survival Guide”, John Wiley & Sons, Inc. 1996.

7. Appendix A: Algorithms for finding the ripple effect in Object-Oriented System

NewTotalEffect()

```
TotalEffect()
// conservertive algorithms to find the ripple of the system
{
  NewSetInit();
  For each class C in ACS
  {
    If C is clean
      continue;
    else mark C clean.
    NewFindEffectInClass(C);
    NewFindEffectAmongChildren(C);
    NewFindEffectAmongClient(C);
  }
}
```

NewInitSys()

```
// define the attribute constant
#define contaminate_none 0x00
#define contaminate_current 0x01
#define contaminate_children 0x02
#define contaminate_client 0x04
#define contaminate_all 0x07

InitSys()
// Initialize the different sets ACS, AMS, AFS etc.
{
  ACS = {set of classes need change};
  For each class C in ACS
  {
    AMS[C] = { methods need change in C}
    AFS[C] = {Data fields need change in C}
  };
  For each class C in ACS
    For each m in AMS[C]
      switch (type of change) {
        case signature:
        case axiom:
        case implementation:
          if m is public method
                m.attribute = contaminate_client | contaminate_children
                        | contaminate_current;
          else if m is protected method
                m.attribute = contaminate_children | contaminate_current;
          else if m is private method
```

```
        m.attribute = contaminate_current;
      break
    case scope:
      if (scope change is public to private)
      // if will affect all of its client and children
            a.attribute = contaminate_children | contaminate_client;
      else if scope is from public to protect
      // it will affect all its clients
        m.attribute = comtaminate_client
      else if (scope is from protect to private)
      // if will affect all of its children
        m.attribute = contaminate_children
      else if ( (scope change is from private to protect)
                  || (scope change is from private to public)
                  || (scope change is from protect to public) )
      // it will not affect any of its clients
        m.attribute = contaminate_none;
        break;
      case add method:
        if m is public method
              m.attribute = contaminate_client | contaminate_children
                    | contaminate_current;
        else if m is protected method
          m.attribute = contaminate_children
               | contaminate_current;
        else if m is private method
          m.attribute = contaminate_current;
          break;
        }
      case delete method:
        if m is public method
              m.attribute = contaminate_client | contaminate_children
                    | contaminate_current;
        else if m is protected method
          m.attribute = contaminate_children
               | contaminate_current;
        else if m is private method
          m.attribute = contaminate_current;
        break;
      default:
    }  // end of switch

// initialize field.
For each f in AFS[C]
switch (change_type) {
  case value changed:
    if f has cause a state change
        if f is public data member
              f.attribute = contaminate_client | contaminate_children
                    | contaminate_current;
        else if f is protected data member
          f.attribute = contaminate_children
               | contaminate_current;
        else if f is private data member
```

```
            f.attribute = contaminate_current;
        else
          f.attribute = contaminate_none;
        break;
      case type change:
            if f is public data member
                  f.attribute = contaminate_client | contaminate_children
                        | contaminate_current;
            else if f is protected data member
              f.attribute = contaminate_children
                   | contaminate_current;
            else if f is private data member
              f.attribute = contaminate_current;
        break
      case scope change:
        if (scope change is public to private)
        //it will affect all of its client and children
          f.attribute = contaminate_client | contaminate_children
        else if scope is from public to protect
        // it will affect all its clients
          f.attribute = comtaminate_client
        else if (scope is from protect to private)
        //if will affect all of its children
            f.attribute = contaminate_children
        else if ((scope change is from private to protect)
                 || (scope change is from private to public)
                 || (scope change is from protect to public) )
          // it will not affect any of its clients
            f.attribute = contaminate_none;
      case add date member:
            if f is public data member
                  f.attribute = contaminate_client | contaminate_children
                        | contaminate_current;
            else if f is protected data member
              f.attribute = contaminate_children
                   | contaminate_current;
            else if f is private data member
              f.attribute = contaminate_current;
        break
      case delete data member:
            if f is public data member
                  f.attribute = contaminate_client | contaminate_children
                        | contaminate_current;
            else if f is protected data member
              f.attribute = contaminate_children
                   | contaminate_current;
            else if f is private data member
              f.attribute = contaminate_current;
        break;
      default:
    }  // end of switch
  }  // end of for each C in ACS
}
```

NewEffectInClass(C)

```
// find the effect within class C
// input: the AMS and AFS in C
// output: the AMS and AFS in C
{
  OLDAMS[C] = AMS[C];
  OLDAFS[C] = AFS[C];
  Analyze the CFG and DFG of each methods, construct MREF & FREF for each methods and data
      fields.

  // process each method in C
  for each method m in class C
    if (( (AffectingFactors = MREF(m) ∩ AMS(C)) ≠ Empty ) or
                ( (AffectingFactors = FREF(m) ∩ AFS(C)) ≠ Empty ))
      If ( ∃factor ∈ AffectingFactors ∩ factor.attribute | contaminate_current ≠ 0 )
      // there exist a  method factor in AffectingFactors set that
       // will affect the data members and methods in current class.
      {
        m.type = indirect;
        AMS(C) = AMS(C) ∪ {m} ;
        if m is public {
           PAMS(C) = PAMS(C) ∪ {m} ;
          m.attribute = contaminate_all;
        } else if m is protected
          m.attribute = contaminate_children
                        | contaminate_current;
        else if m is private
          m.attribute = contamiate_current;
      }

  // Process each data member in class C
  For each data member f in C
    if (( (AffectingFactors = FREF(f) ∩ AFS(C)) ≠ Empty ) or
                ( (AffentingFactors = FREF(f) ∩ AFS(C)) ≠ Empty ))
      If ( ∃factor ∈ AffectingFactors ∩ factor.attribute | contaminate_current ≠ 0 )
      // there exist a  data member in AffectingFactors set that
       // will affect the data members and methods in current class.
      {
        f.type = indirect;
         AFS(C) = AFS(C) ∪ {f} ;
        if f is public {
            PAFS(C) = PAFS(C) ∪ {f} ;
          f.attribute = contaminate_all;
        } else if f is protected
          f.attribute = contaminate_children
                        | contaminate_current;
        else if f is private
          f.attribute = contamiate_current;
      }

  if ( (OLDAMS[C] ≠ AMS[C]) or (OLDAFS[C] ≠ AFS[C]) )
  {
     ACS = ACS ∪ {C} ;
```

```
        mark C dirty;
      }
    }
```

NewEffectAmongChildren()

NewFindEffectAmongChildren(C_p)
// find the effect of all the children, when the parent changes
{
For each class C_c which inherited from C_p
  For each method m in C_c {
  switch (inheritance type of m) {
  case m is New:
  case m is Virtual New:
  // Since m is new, the change in parent will not affect it.
    break;
  case Extended Redefine:
  // still use parent's service
    if ($m_p \in C_p \cap m_p \in AMS[C_p] \cap m_p$.attribute & contaminate_children $\neq 0$) {
      $AMS(C_c) = AMS(C_c) \cup \{m\}$
    if m is public {
      $PAMS(C) = PAMS(C) \cup \{m\}$;
      m.attribute = contaminate_all;
    } else if m is protected
      m.attribute = contaminate_children
                  | contaminate_current;
    else if m is private
      m.attribute = contamiate_current;
    }
    break;
  case Total Redefine:
  // m does not use the service of A in parent. It will not be affected by the change of m in A,
    break;
  case virtual Extended Redefine:
  // m extended the function of m in parent by using the service of m in parent. So the change of m in parent
    if ($m_p \in C_p \cap m_p \in AMS[C_p] \cap m_p$.attribute & contaminate_children $\neq 0$) {
      $AMS(C_c) = AMS(C_c) \cup \{m\}$
    if m is public {
      $PAMS(C) = PAMS(C) \cup \{m\}$;
      m.attribute = contaminate_all;
    } else if m is protected
      m.attribute = contaminate_children
                  | contaminate_current;
    else if m is private
      m.attribute = contamiate_current;
    }
    break;
  case Virtual Total Redefine:
  // Even though m totally redefine the implementation of m in parent. If the m in

// parent's signature has changed, it will affect the child, according to
// Liskov substitution rule.
if ($m_p \in C_p \cap m_p \in AMS[C_p] \cap m_p$.attribute & contaminate_children $\neq 0$) {
$AMS(C_c) = AMS(C_c) \cup \{m\}$
```
if m is public {
```
$PAMS(C) = PAMS(C) \cup \{m\}$;
```
m.attribute = contaminate_all;
} else if m is protected
  m.attribute = contaminate_children
                | contaminate_current;
else if m is private
  m.attribute = contamiate_current;
```
}
break;
case inherit, virtual inherit
if ($m_p \in C_p \cap m_p \in AMS[C_p] \cap m_p$.attribute & contaminate_children $\neq 0$) {
$AMS(C_c) = AMS(C_c) \cup \{m\}$
```
if m is public {
```
$PAMS(C) = PAMS(C) \cup \{m\}$;
```
  m.attribute = contaminate_all;
} else if m is protected
  m.attribute = contaminate_children
                | contaminate_current;
else if m is private
  m.attribute = contamiate_current;
```
}
break;
default:
break;
}
// if m is in AMS set already, check next item
if ($m \in AMS[C_c]$)
continue;
// no matter m in affected by the m in parent, if it use any other methods in AMS
// or any data fields in AFS of parent, it will be affected.
if (($AffectingFactors = MREF(m) \cap AMS(C_p)) \neq Empty$) or
(($AffectingFactors = FREF(m) \cap AFS(C_p)) \neq Empty$))
If ($\exists factor \in AffectingFactors \cap$ factor.attribute | contaminate_children $\neq 0$) {
$AFS(c) = AFS(c) \cup \{m\}$
```
if m is public {
```
$PAMS(C) = PAMS(C) \cup \{m\}$;
```
  m.attribute = contaminate_all;
} else if m is protected
  m.attribute = contaminate_children
                | contaminate_current;
else if m is private
  m.attribute = contamiate_current;
```
}
end // end of for each method
end // end of for each class

}

NewEffectAmongClients()

```
NewEffectAmongClients (C0)
// input:  C0 and its PAMS, PAFS.
// output: All the affected classes that use C0, and their AMS, AFS, PAMS, PAFS
For each class C that uses C0
{
  OLDAMS[C] = AMS[C];
  OLDAFS[C] = AFS[C];
  For each methods m in C {
      if ( ( (AffectingFactors = MREF(m) ∩ AMS(C)) ≠ Empty ) or
             ( (AffectingFactors = FREF(m) ∩ AFS(C)) ≠ Empty ) )
        if ( ∃factor ∈ AffectingFactors ∩ factor.attribute | contaminate_client ≠ 0 ) {

            AMS(C) = AMS(C) ∪ {m}
            if m is public {
               PAMS(C) = PAMS(C) ∪ {m} ;
              m.attribute = contaminate_all;
            } else if m is protected
              m.attribute = contaminate_children
                        | contaminate_current;
            else if m is private
              m.attribute = contamiate_current;
        } // end of if
  } // end of for each method
  For each field f in C {
    if (( (AffectingFactors = FREF(f) ∩ AFS(C)) ≠ Empty ) or
              ( (AffentingFactors = FREF(f) ∩ AFS(C)) ≠ Empty ) )
      If ( ∃factor ∈ AffectingFactors ∩ factor.attribute | contaminate_client ≠ 0 )
      // there exist a  data member in AffectingFactors set that
       // will affect the data members and methods in current class.
      {
        f.type = indirect;
         AFS(C) = AFS(C) ∪ {f} ;
        if f is public {
           PAFS(C) = PAFS(C) ∪ {f} ;
          f.attribute = contaminate_all;
        } else if f is protected
          f.attribute = contaminate_children
                        | contaminate_current;
        else if f is private
          f.attribute = contamiate_current;
      } //end of if
  } // end of each field
  EffectInClass(C)
  if ( (OLDAMS[C] ≠ AMS[C]) or (OLDAFS[C] ≠ AFS[C]) )
  {
     ACS = ACS ∪ {C} ;
    mark C dirty;
  }
  } // end of each class
}
```